基于Python的自动化代码审计

逢魔安全实验室 - xfkxfk

自我介绍

```
~ $ id
uid=0(xfkxfk@formsec)
gid=0(web安全研究员，工控安全研究员)
groups=0(FormSec)
```

内容简介

Python常规漏洞

常规漏洞

SQL注入

```
def  getuser(request):
    username = request.POST.get('username')
    query = 'select * from users where username=%s'%username

    connection = psycopg2.connect(dbname,user,host,password)
    curs = connection.cursor()
    curs.execute(query)
    res = curs.fetchall()
    connection.clouse()

    return res
```

常规漏洞

注入防御

```
def  getuser(request):
    username = request.POST.get('username')
    query = 'select * from users where username=%s'

    connection = psycopg2.connect(dbname,user,host,password)
    curs = connection.cursor()
    curs.execute(query, [username])
    res = curs.fetchall()
    connection.clouse()

    return res
```

常规漏洞

命令执行

```
def  store_uploaded_file(request):
    uploaded_file = request.POST.get('filename')
    upload_dir_path = "static/uploads  “

    if not os.path.exists(upload_dir_path):
        os.makedirs(upload_dir_path)

    cmd = "mv" + uploaded_file + "" + "%s" % upload_dir_path
    os.system(cmd)

    return 'static/uploads/%s'  % uploaded_file
```

常规漏洞

执行防御

```
def  store_uploaded_file(request):
    uploaded_file = request.POST.get('filename')
    upload_dir_path = "static/uploads"

    if not os.path.exists(upload_dir_path):
        os.makedirs(upload_dir_path)

    cmd = "mv" + uploaded_file + "" + "%s" % upload_dir_path
    subprocess.Popen(cmd, shell=False)

    return 'static/uploads/%s'  % uploaded_file
```

其他漏洞……

- XSS
- XXE
- CSRF
- SSRF
- SSTI
- 代码注入
- 目录穿越
- 越权操作
- 其他...

静态分析

备注

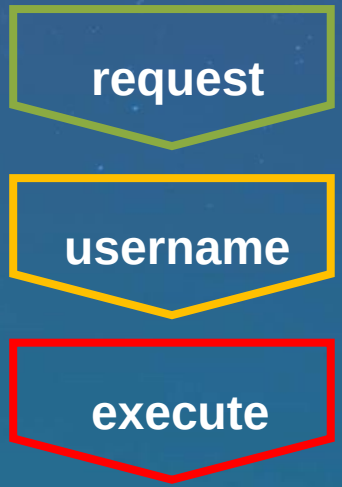
从python常规漏洞来看都有一个共同点，那就是危险函数中使用了可控参数，
如system函数中使用到的(‘mv %s’% filename),
如execute函数中使用到的username参数，
如HttpResponse中使用到的nickname参数，

这些参数直接从第一层入口函数中传进来，或者经过简单的编码，
截断等处理直接进入危险函数，导致了以上危险行为。

静态分析的核心是什么？

静态分析

```
from taskMnager.forms import PorjectFileForm

def getproj(request, project_id):
    if request.mothod = “POST”:
        username = request.POST.get(‘name’, False)
        curs = connection.cursor()
        sql = “select * from  pro where name = ‘%s’” % username
        curs.execute(sql)
    else:
        form = ProjectFileForm()

return render(request, ‘index.html’, {‘form’: form})
```



备注

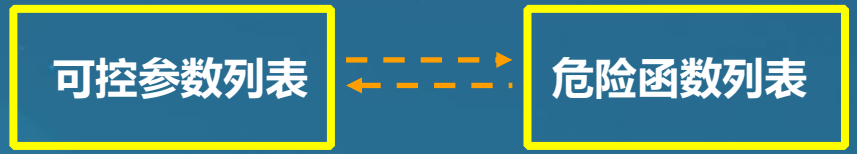
注入判断的核心就在于找到危险函数，并且判断其参数是可控的。

找到危险函数这个只需要维护一个危险函数列表即可，

当在语法树中发现了函数调用并且其名称在危险列表中就可以标记出该行代码，
接下来的难点就在于跟踪该函数的参数，
默认认为该危险函数的外层函数的参数是可控的，
那就只需要分析这个外层函数参数的传递过程即可

静态分析

可控数据（参数）通过系统处理最后进入危险函数



静态分析

查找可控参数

新的变量 = 初始参数经过一系列处理后

新变量可控

直接赋值　属性赋值　字符串拼接

函数处理　分片取值　列表解析式

字符串操作函数　未过滤函数

备注

直接赋值：GET参数直接赋值
属性赋值：request.POST.get(‘name’)赋值，排除META中的内容
字符串拼接：字符串拼接
列表解析式：
元组、列表、字典数据处理：元素相加，赋值value等
Subscript分片取值：通过下标索引取值
函数调用后赋值：字符串操作的系统函数str，strip，split，encode等，未过滤的自定义函数，危险函数

With操作：
For循环：
If判断：

排除特殊情况：
判断是否合法：os.path.exitst，isdir等
锁定范围：Type in [xxx,xxx]

静态分析

将py文件通过ast解析为tree并递归解析导入模块

{

‘body’ : **[{}, {}, {}, …]**,

‘filename’ : ‘/django.nV/taskManager/views.py’ ,

‘type’ : ‘Module’

}

静态分析

获取body中的函数体内容

```
from taskMnager.forms import PorjectFileForm

def getproj(request, project_id):
    if request.mothod = “POST”:
        name = request.POST.get(‘name’, False)
        curs = connection.cursor()
        sql = “select * from  pro where name = ‘%s’” % name
        curs.execute(sql)
    else:
        form = ProjectFileForm()

return render(request, ‘index.html’, {‘form’: form})
```

Body[0]

Body[1]

```
{
  ‘_fields’  :[],
  ‘args’  :{request, project_id},
  ‘body’   : [{}, {}],
  ‘decorator_list’  :[],
  ‘kwarg_name’  :None,
  ‘lineno’  :7,
  ‘name’  :’  getproj’  ,
  ‘name_node’  :{},
  ‘type’  :  ‘FunctionDef’
  ‘vararg_name’  :None
}
```

静态分析

递归body中元素具体内容并查找可控参数

```
from taskMnager.forms import PorjectFileForm

def getproj(request, project_id):
    if request.mothod = “POST”:
        name = request.POST.get(‘name’, False)
        curs = connection.cursor()
        sql = “select * from  pro where name = ‘%s’” % name
        curs.execute(sql)
    else:
        form = ProjectFileForm()

return render(request, ‘index.html’, {‘form’: form})
```

test

body

orelse

```
{
  ‘args’   : None,
  ‘body’   : [],
  ‘orelse’  : [],
  ‘test’   : {},
  ‘lineno’  : 8,
  ‘handlers’  : None,
  ‘type’   :   ‘If’
   ......
}
```

静态分析


```
{'value':
    {
        'starargs': None,
        'args': [{'s': "'name'", 'lineno': 29, 'type': 'Str'}, {'lineno': 29, 'type': 'Name', 'id': 'False'}],
        'lineno': 29,
        'func':
            {
                'attr': 'get',
                'value':
                    {
                        'attr': 'POST',
                        'value': {'lineno': 29, 'type': 'Name', 'id': 'request'},
                        'lineno': 29,
                        '_fields': ['value', 'attr_name'],
                        'type': 'Attribute'
                    },
                'lineno': 29,
                '_fields': ['value', 'attr_name'],
                'type': 'Attribute'
            },
        'kwargs': None,
        'keywords': [],
        'type': 'Call'
    },
    'lineno': 29,
    'type': 'Assign',
    'targets': [{'lineno': 29, 'type': 'Name', 'id': 'name'}]
}
```


```
name = request.POST.get('name', False)
```

静态分析

判断漏洞流程

备注

如果存在此文件中导入了其他非系统模块，继续递归解析此模块文件
如果存在类的话，继续递归类里面方法的内容
Body的内容是嵌套的，一个body里面可能还有很多个body

循环body体中的元素，然后取出body中的body，orelse，test，handlers元素，继续递归查找可控参数

以行为单位解析出来的结构和内容
Name为被赋值的变量名
然后value里面就是具体的内容
从右往左一次嵌套，所以request在最里层的value

以Python文件为入口，解析成语法树，格式化为json格式
取出语法树中的函数体内容
然后遍历函数体中的代码行：
如果有危险函数调用，并且有可控参数进入此危险函数，则报出漏洞

所以这里的核心就是：
1、递归全部代码查找可控参数，生成可控参数列表
2、维护危险函数列表

静态分析

- 弃用正则
- 方便使用
- 易于扩展
- 易于维护

- 漏报误报
- 外部导入的影响

https://github.com/xfkxfk/pyvulhunter

动态分析

动态分析

```
$  cd /test
$  ls
string.py
$  cat string.py
def  upper(s):
            return   “I LOVE FORMSEC”

$  PYTHONPATH = /test
$  python
Python 2.7.6 (default, Oct 26 2016, 20:32:47)
[GCC 4.8.4] on linux2
>>>  import   string
>>>  string.upper( “I  love  ssc” )  #I LOVE SSC
>>>    “I LOVE FORMSEC”
```

动态分析

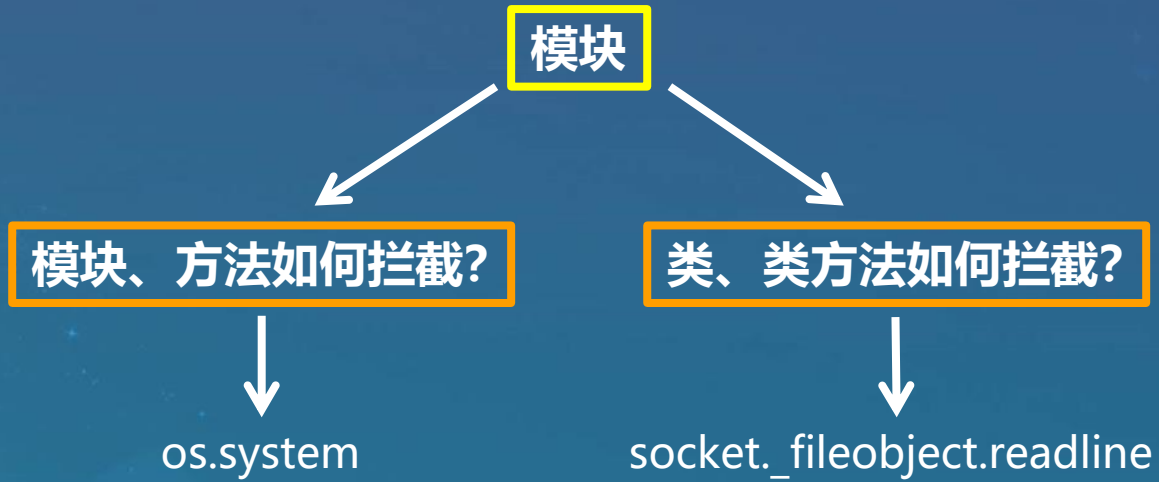
PYTHONPATH

- Built-in modules
- Sys.path
  当前工作目录
  系统变量$PYTHONPATH

NAMESPACE

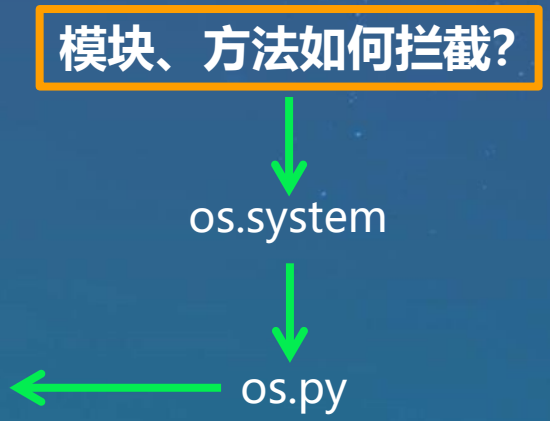
- 命名空间的开放性
- 覆盖任意模块或者定义的命名空间
- **劫持函数方法，拦截参数内容**

动态分析


模块
模块、方法如何拦截?
类、类方法如何拦截?
os.system
socket._fileobject.readline

动态分析

```
Class _InstallFcnHook(object):
    def __init__(self, fcn):
        self._fcn = fcn

    def _pre_hook(self, *args, **kwargs):
        print "[*] Insecure command found!"
        return (args, kwargs)

    def __call__(self, *args, **kwargs):
        (_hook_args, _hook_kwargs) = self._pre_hook(*args, **kwargs)
        retval = self.fcn(*_hook_args, *_hook_kwargs)
        return retval

fd, pathname, desc = imp.find_module(__name__, sys.path[::-1])
mod = imp.load_module(__name__, fd, pathname, desc)

system = _InstallFcnHook(system)
```



动态分析

```
Class _InstallFcnHook(object):
    def __init__(self, fcn):
        self._fcn = fcn

    def _pre_hook(self, *args, **kwargs):
        print “[*] Insecure command found!”
        return (args, kwargs)

    def __call__(self, *args, **kwargs):
        (_hook_args, _hook_kwargs) = self._pre_hook(*args, **kwargs)
        retval = self.fcn(*_hook_args, *_hook_kwargs)
        return retval

fd, pathname, desc = imp.find_module(__name__, sys.path[::-1])
mod = imp.load_module(__name__, fd, pathname, desc)

system = _InstallFcnHook(system)
```

```
$ mv  ./os.py  /tmp
$ cd  /tmp
$ PYTHONPATH=/tmp  python
>>> import  os
>>> os.system('whoami;pwd')
[*] Insecure command found!
xfkxfk
/tmp
0
>>>
```

备注

python 是一种动态类型语言，python 中一切皆对象
所以换句话说每个对象可以在程序里任何地方改变它

这就意味着我们可以劫持我们认为危险的函数
拦截进入函数的参数，判断是否有恶意参数进入，从而判断是否存在漏洞

Python的广泛使用，很大部分是因为开发效率高，模块使用方便
所以就劫持就针对：
1、模块的直接方法
2、模块的类，已经类方法进行了

举例：模块的方法可以直接被劫持
首先通过imp导入os模块，然后在覆盖到其中的system方法
在调用system方法时，就是这里的__call__方法了
判断进入system方法的参数是否有恶意内容，从而可以判断是否真正触发了漏洞

动态分析

类、类方法如何拦截?

- 元类
- type
- __metaclass__
- __new__
- __call__
- __getattribute__

备注

元类：
元类就是用来创建类的类，函数type实际上是一个元类
元类的主要目的就是为了当创建类时能够自动地改变类。

__metaclass__：
你可以在写一个类的时候为其添加__metaclass__属性，Python就会用它来创建类
__metaclass__可以接受任何可调用的对象，你可以在__metaclass__中放置可以创建一个类的东西
__new__:是用来创建类并返回这个类的实例
__call__:任何类只需要定义一个__call__()方法，就可以直接对实例进行调用,用callable来判断是否可被调用
__getattribute__:定义了你的属性被访问时的行为

你首先写下class Foo(object)，但是类对象Foo还没有在内存中创建。
Python会在类的定义中寻找__metaclass__属性，如果找到了，Python就会用它来创建类Foo，
如果没有找到，就会用内建的type来创建这个类

动态分析

```
# __metaclass__ = UpperAttr

Class Test(object):
        __metaclass__ = UpperAttr

        def foo(self, name):
                pass
```

Test类所有属性名称大写?

```
Class UpperAttr(type):
        #create and return class instance
        def __new__(meta, name, bases, dct):
                #
                # upper attribute name here
                #
                return type.__new__(meta, name, bases, dct)
```

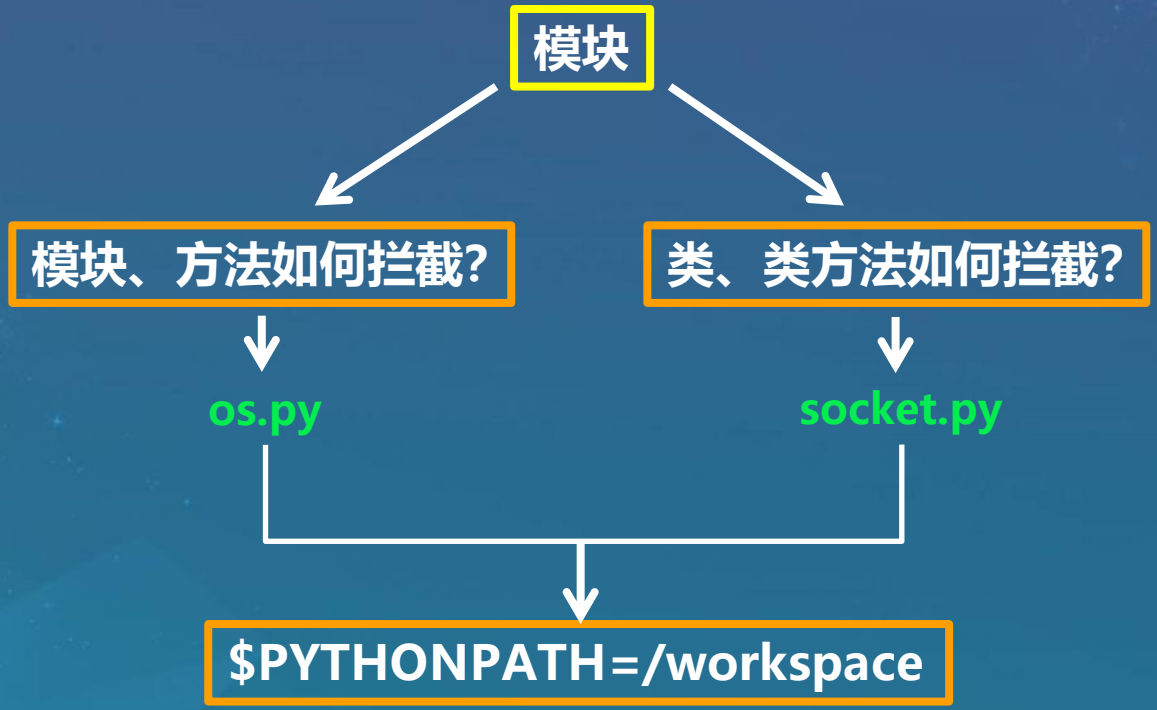
动态分析

```
# __metaclass__ = UpperAttr

Class _fileboject(object):
        __metaclass__ = _InstallClsHook

        def _hook_readline(self, name):
                #do something here

        def _hook_writeline(self, name):
                #do something here
```

```
Class _InstallClsHook(type):
        #create and return class instance
        def __new__(meta, name, bases, dct):
                bases = (GetAttribute,)
                return type.__new__(meta, name, bases, dct)
```

```
Class GetAttribute:
        def __ getattribute __(self, name):
                retval = object.__getattribute__(self, method)
                if callable(retval):
                        return object.__getattribute__(self,
  '_hook_'  + method)
```

动态分析



动态分析

__builtin__ : open?

Monkey patch

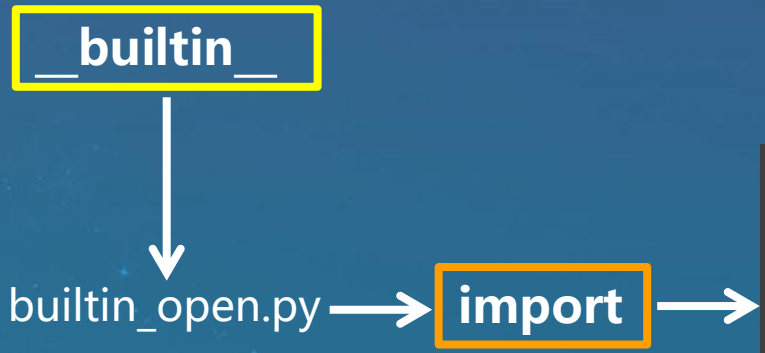
- module: 模块名
- function: 要hook的函数
- action: 处理拦截的内容
- hook_arg: 判断拦截输入参数
- hook_result: 判断拦截输出数据

动态分析

```
def hook_func(module, function, action, hook_arg, hook_result):
        old_module = __import__(module)
        old_function = getattr(old_module, functic

        def wrapper(*args, **kwargs):
                if hook_arg:
                        action(values)
                if hook_result:
                        action(result)

        setattr(old_module, function, wrapper)

def action(values):
        #do somethin to judge vuln

if __name__ == "__main__":
        hook_func("__builtinin_i_", "open", action, hook_arg, hook_result)
        open( "../../../../../etc/passwd" )
```

```
$ ls
builtin_open.py
$ python builtin_open.py

***Insecure filepath found!***
Traceback (most recent call last):
  File "builtin_open.py", line 40, in <module>
    open('../../../../../etc/passwd')
  File "builtin_open.py", line 15, in wrapper
    act(values)
  File "builtin_open.py", line 33, in check_file_path
    raise Exception('Insecure filepath found!')
Exception: Insecure filepath found!
$
```

动态分析




```
#Views.py

from builtin_open import hook_func

@hook_func
def download(request, project_id):
        #download file at here
```

动态分析

- 动态检测
- 准确性高
- 易于维护
- 可平台化

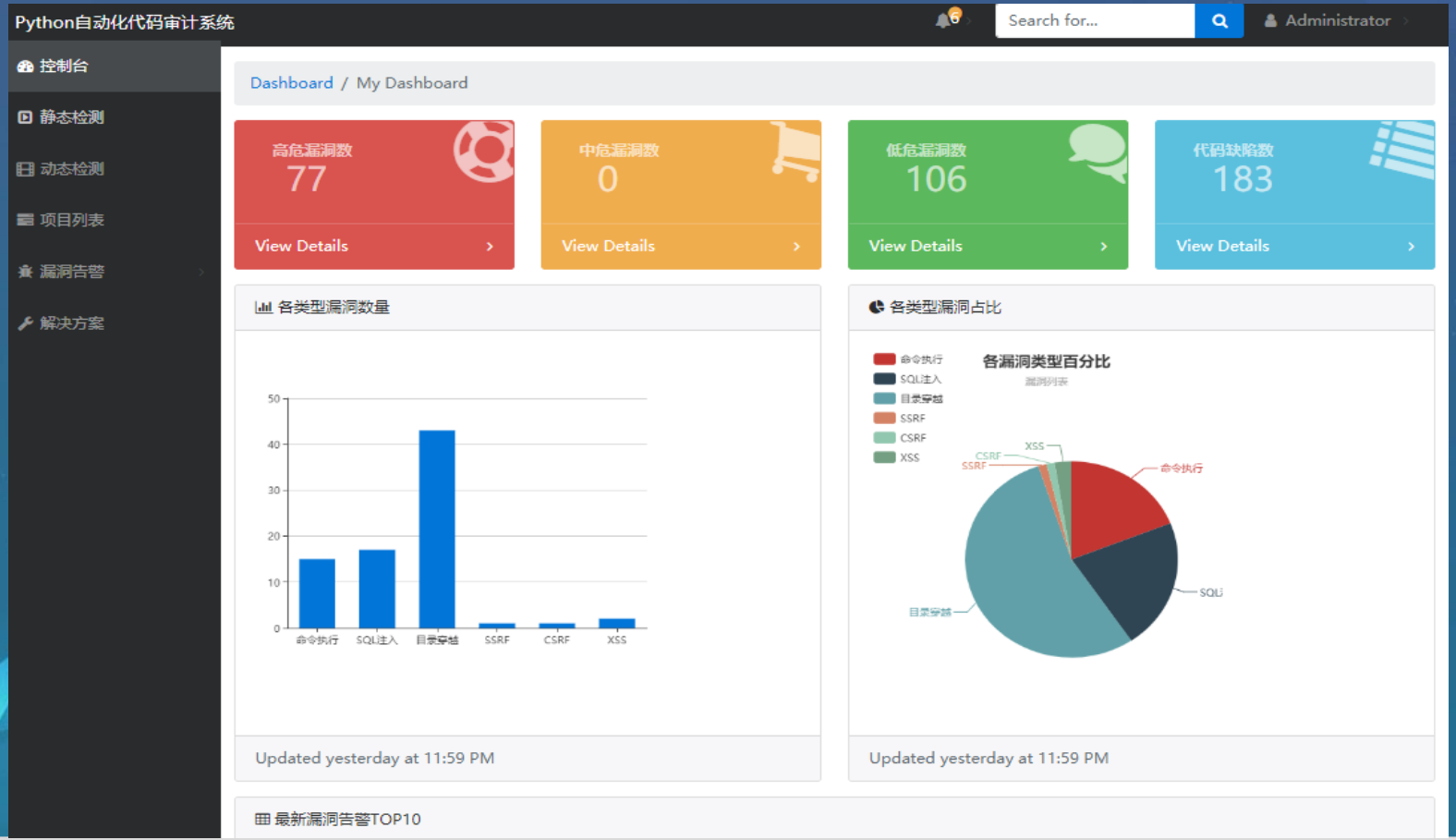
- 扩展门槛
- 对第三方接口无效
- 对系统自身的影响

https://github.com/xfkxfk/pyekaboo

自动化应用

自动化应用

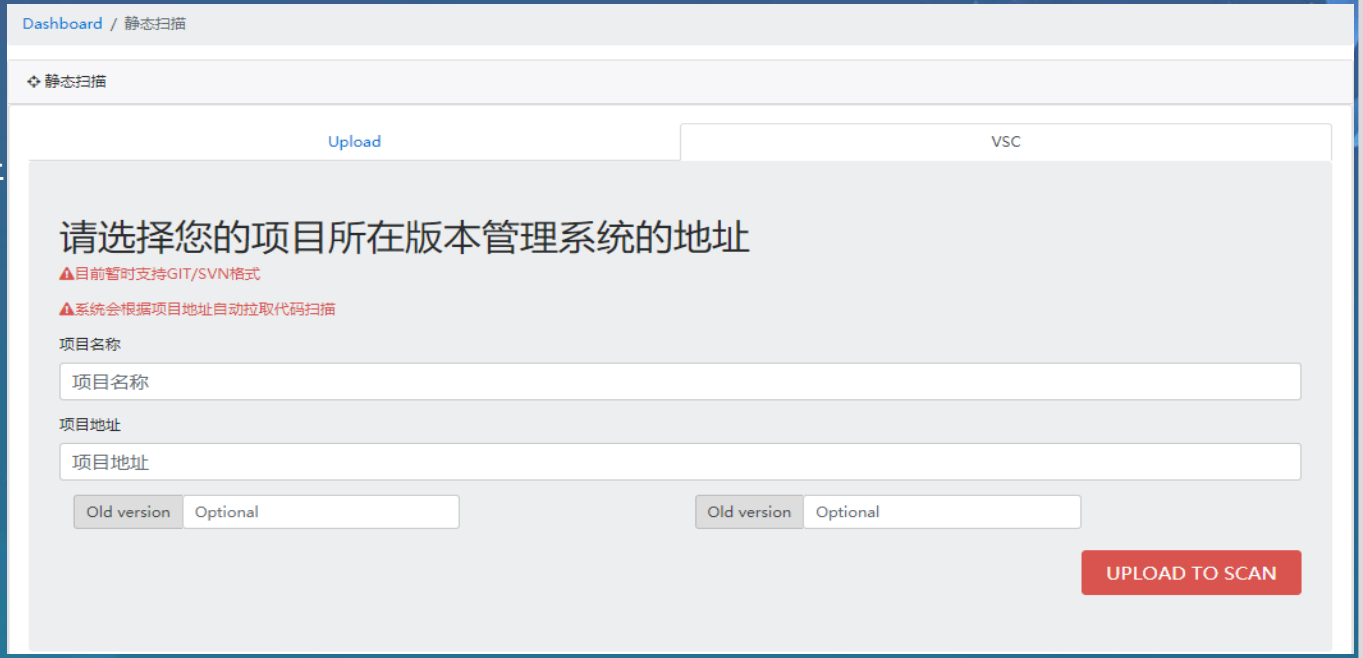
自动化应用


Python自动化代码审计系统
Search for...
Administrator
控制台
静态检测
动态检测
项目列表
漏洞告警
解决方案
Dashboard / My Dashboard
高危漏洞数
77
View Details
中危漏洞数
0
View Details
低危漏洞数
106
View Details
代码缺陷数
183
View Details
各类型漏洞数量
50
40
30
20
10
0
命令执行
SQL注入
目录穿越
SSRF
CSRF
XSS
Updated yesterday at 11:59 PM
各类型漏洞占比
各漏洞类型百分比
命令执行
SQL注入
目录穿越
SSRF
CSRF
XSS
Updated yesterday at 11:59 PM
最新漏洞告警TOP10

自动化应用

静态检测应用

- 上传代码打包文件
- 对接git服务器
- 对接svn服务器



自动化应用

静态检测应用

- 静态漏洞详情

漏洞详情

漏洞名称：命令执行漏洞

扫描方式：staticscan

危险等级：high

漏洞文件：`/django.nV-master/taskManager/misc.py`

调用链：`store_uploaded_file--->set([u'os.system'])`

漏洞所在行数：33

函数名称：`os.system`

漏洞代码：

```
def store_uploaded_file(title, uploaded_file):
    """ Stores a temporary uploaded file on disk """
    upload_dir_path = '%s/static/taskManager/uploads' % (
        os.path.dirname(os.path.realpath(__file__)))
    if not os.path.exists(upload_dir_path):
        os.makedirs(upload_dir_path)

    # A1: Injection (shell)
    # Let's avoid the file corruption race condition!
    os.system(
        "mv " +
```

漏洞描述：

在用户输入的字符串之中注入一些隔断符加上其他系统指令，由于设计缺陷在程序当中忽略了检查，这些注入进去的恶意隔断符和系统指令被系统正常执行，因此服务器或者应用程序遭到破坏或者攻击。

漏洞修复：

1、尽量不用原生的shell命令；

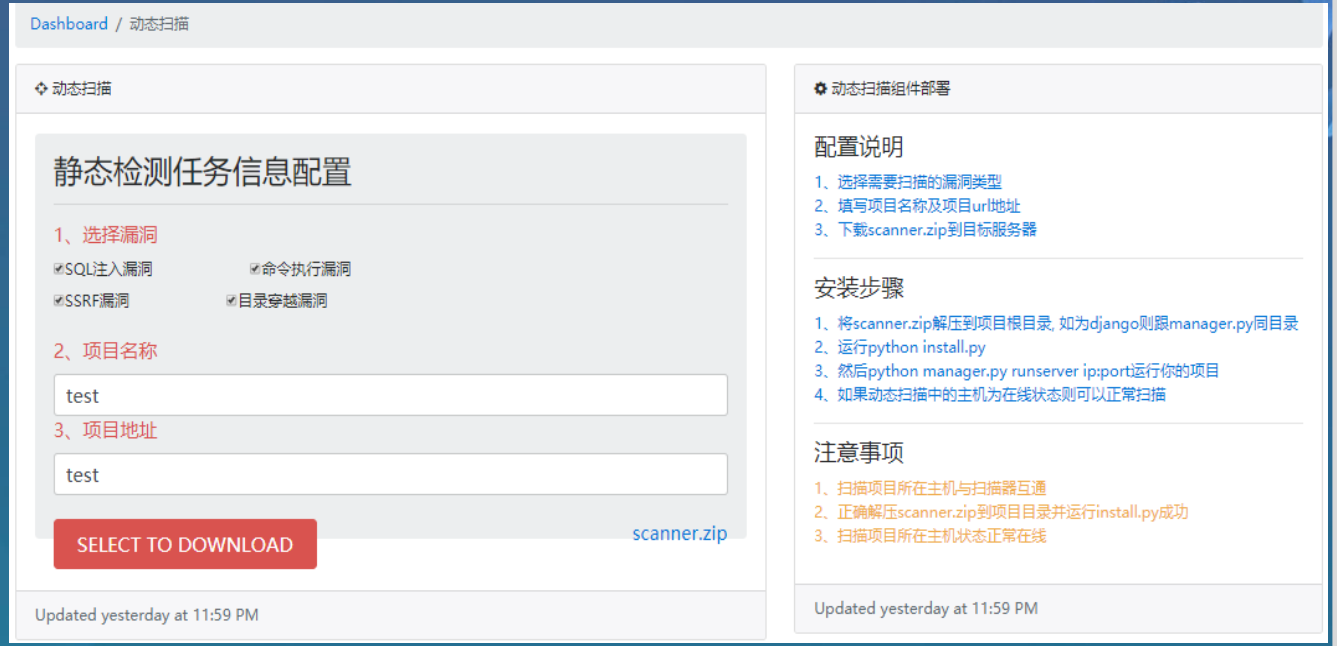
2、封装命令执行API，过滤所有shell元字符（&，|，$....）；

3、在用户输入入口做好严格过滤。

自动化应用

动态检测应用

- 选择要检测的漏洞
- 下载agent安装包
- 部署agent到服务器



自动化应用

动态检测应用

- 成功部署agent
- 主机在线列表

在线主机列表

Show 10 entries

Search:

ID	Pro Name	Hostname	IP	Time	Status
26	django.nV	-test		17-07-07	在线
24	test			17-07-10	在线
ID	Pro Name	Hostname	IP	Time	Status

Showing 1 to 2 of 2 entries

Previous 1 Next

自动化应用

动态检测应用

- 设置代理
- 正常访问
- 自动检测

自动化应用

动态检测应用

- 动态检测漏洞详情

漏洞详情

漏洞名称：目录穿越漏洞

扫描方式：dynamicscan

危险等级：high

请求详情：

```
METHOD: GET
URL: http://10.65.20.192:8001/taskManager/download/22/
GETPARAMS:
REFERER: http://10.65.20.192:8001/taskManager/1/
CONNECTION: keep-alive
HOST: 10.65.20.192:8001
ACCEPT: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
ACCEPT_LANGUAGE: zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3
COOKIE: csrftoken=Kboe2EVJi9yLJ7pEKD5Mq0y7mn4uHcOZJ47Bgc31dHpRJPsf49sirLXbw5zKOv6N; sessionid=".eJxNzLEOgjAUQNGCOhoTvwKXRgpUWd0dm7A1bd8roKZNKR1N_HQ1YWA_937yd8
USER_AGENT: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:54.0) Gecko/20100101 Firefox/54.0
UPGRADE_INSECURE_REQUESTS: 1
ACCEPT_ENCODING: gzip, deflate
```

漏洞文件：`/home/android/django.nV/taskManager/views.py`

污点参数值：`[u'/home/android/django.nV/taskManager/static/taskManager/uploads/../../../../../../../../../../etc/passwd', u'rb']`

漏洞所在行数：219

函数名称：`download`

调用链：`download` 点击查看详情

漏洞代码：

```
filename: /home/android/django.nV/taskManager/views.py
def download(request, file_id, file_name="test"):
    file = File.objects.get(pk=file_id)
    abspath = open(
        os.path.dirname(
            os.path.realpath(__file__)) +
        file.path,
        'rb')
    response = HttpResponse(content=abspath.read())
    response['Content-Type'] = mimetypes.guess_type(file.path)[0]
    response['Content-Disposition'] = 'attachment; filename=%s' % file.name
    return response
```

漏洞描述：

在用户输入的字符串之中注入../等目录回溯字符串，由于设计缺陷在程序当中忽略了检查，导致程序中路径参数可控，可穿越到上级目录，最终导致任意文件删除，下载，覆盖等攻击。

漏洞修复：

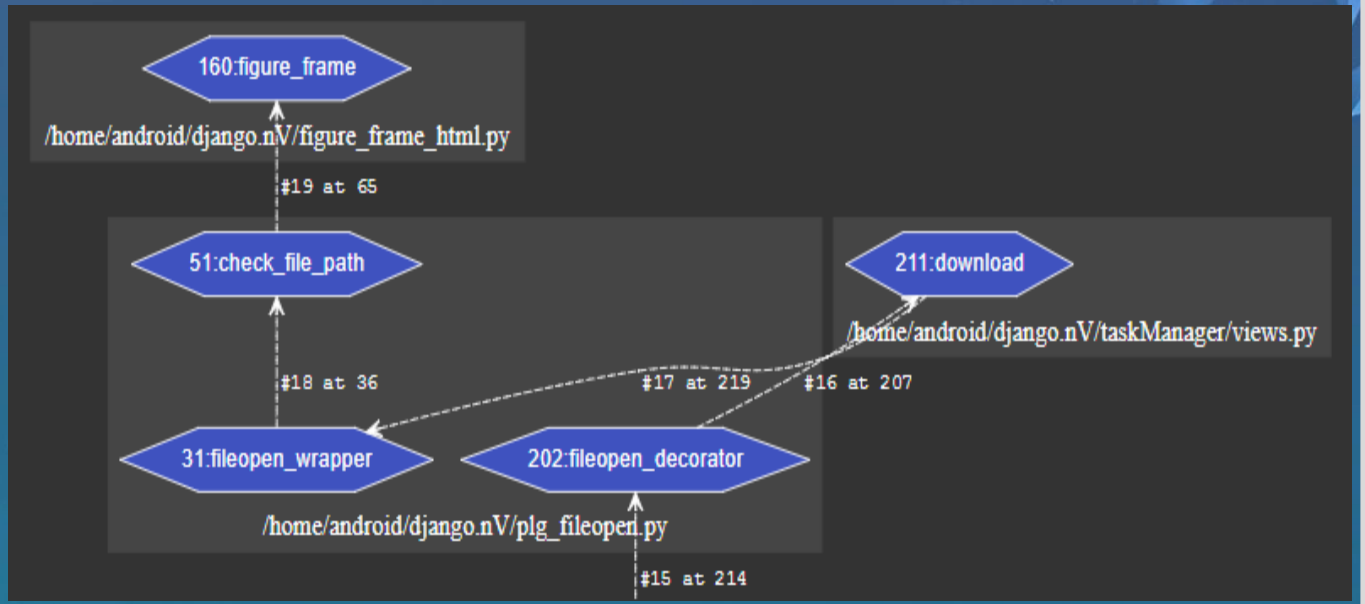
1、严格验证参数范围，非法字符直接返回；

2、过滤..和./字符，但是不要直接过滤../。

自动化应用

动态检测应用

- 动态检测漏洞详情
- 动态检测调用栈



自动化应用

Do What?

- 节省项目上线时安全测试资源
- 及时发现项目开发过程安全问题
- 提高开发者安全开发意识
- **提高项目安全性**

自动化应用

To Do

- 增强静态检测准确性
- 丰富动态扫描漏洞类型
- 减少动态扫描对系统代码的变动
- 模块化导入直接使用
- 开源?

感谢观看！